第3章

视频讲解

Spring Boot的核心概念

在第2章的案例中，将业务逻辑集中在控制器层面对于小型项目来说是一种快速实现的方法，但随着项目规模的增长，这种方法可能会导致控制器代码变得庞大和难以维护，进而造成结构上的混乱。因此，本章将介绍分层架构的设计原则，目的是将复杂的业务逻辑从控制器中分离出来，转移到专门的服务类中去处理。这种转变有助于使控制器回归其核心功能——处理路由和渲染视图，同时提高代码的清晰度和可维护性。

3.1 三层架构

三层架构是一种常见的软件系统设计模式，它将应用划分为三个关键层次：表现层、业务逻辑层和数据访问层。表现层专注于用户界面的展示和用户交互。业务逻辑层作为核心，处理应用的功能实现和业务规则。数据访问层则与数据存储系统直接交互，执行数据的读取、写入和更新。这种设计不仅增强了代码的可维护性和可扩展性，还通过减少各层间的依赖，提升了系统的灵活性和重用性。

在Web应用中，三层结构如图3-1所示。控制器和视图共同构成表现层，负责处理用户界面

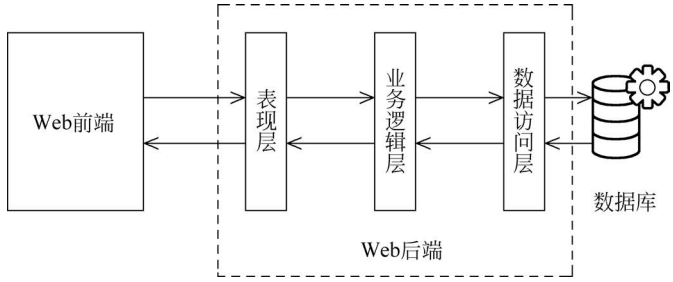

图3-1　三层结构

的展示和交互。服务层和业务层紧密结合,形成业务逻辑层,专注于实现应用的核心功能和业务规则。而存储库或数据访问对象(DAO)则构成数据访问层,专门负责与数据库的交互,执行数据的存取操作。这种分层方法不仅明确了各层的职责,还促进了团队成员之间的高效协作,提升了开发效率和项目的可维护性。

3.1.1 表现层

在 Spring Boot 的应用中,表现层负责接收和响应 HTTP 请求,同时构建用户界面以实现与用户的交互。这一层通常借助 Spring MVC 或 Spring WebFlux 框架来实现,通过注解定义控制器,将 HTTP 请求映射到特定的方法,然后返回数据或视图给客户端。为了创建动态的用户界面,开发者可以选择 Thymeleaf、Freemarker 等服务器端模板引擎,或者结合 React、Vue.js 等现代前端框架,以提供更加丰富和交互式的用户体验。

在第 1 章的综合案例中,项目利用 Spring Boot 的 Spring MVC 组件构建表现层。控制器负责接收 HTTP 请求,通过与 Thymeleaf 模板引擎的结合,动态生成 HTML 内容,有效地展示文章列表和文章详情,从而为用户提供了直观的阅读体验。

3.1.2 业务逻辑层

业务逻辑层负责处理应用程序的业务规则和逻辑。在这一层,定义了多个服务组件,每个服务组件承担特定的业务逻辑任务,如数据操作、规则验证和业务流程的实现。将业务逻辑封装进服务中,不仅促进了代码的模块化,还提高了代码的可重用性。

【例 3-1】 将例 2-1 的业务逻辑拆分到单独的服务类中。

创建 PostService 类以封装博客文章业务逻辑,并通过在 PostController 中调用其方法来处理请求。调整后的代码如下。

(1) PostService 类。

```
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
public class PostService {
    private static final List<Post> posts = new ArrayList<>();

    static {
        // 静态初始化块填充预定义的文章数据
        posts.add(new Post(1L, "欢迎光临我的博客!", "这是第一个帖子",
                "张三", LocalDateTime.of(2029, 7, 1, 12, 0)));
        posts.add(new Post(2L, "如何学习 Spring Boot", "通过 AI 工具学习是个好办法",
                "李四", LocalDateTime.of(2029, 7, 15, 14, 30)));
        // 添加更多预定义文章...

    }

    public List<Post> getAllPosts() {
        return posts;
```

```
    }
}
```

PostService 类专门用于封装博客文章的所有业务操作。

(2) PostController 类。

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService postService = new PostService();
    @GetMapping
    public ResponseEntity<List<Post>> getAllPost() {
        // 返回文章列表
        return ResponseEntity.ok()
                .body(postService.getAllPosts());
    }
}
```

在 PostController 类中，控制器会委托 PostService 类来执行相应的业务逻辑。通过这种设计，简化了控制器的职责，使其专注于处理 HTTP 请求和响应。将业务逻辑迁移至服务层，实现了分层架构，明确了各层职责。这种架构优化了控制器的简洁性，同时将业务逻辑封装在服务层，增强了代码的可维护性和测试性。

在 PostController 类中，我们直接实例化了 PostService 对象，并将其赋值给成员变量 postService。虽然这初步实现了业务逻辑与控制层的分离，但并未充分利用现代框架如 Spring 推荐的依赖注入(DI)的优势。在 3.2.1 节将详细讲解依赖注入的核心思想和优势。

3.1.3　数据访问层

在上述示例中，PostService 类同时承担了业务逻辑和数据访问的角色，导致代码耦合度增加，影响了可维护性。为了提高架构的清晰度，应将数据访问逻辑独立出来，创建一个专门的数据访问层。这样，服务层可以专注于业务逻辑，而数据访问的具体实现则由数据访问层负责。这种分离策略简化了代码结构，提升了系统的可维护性和测试性。

【例 3-2】 为了进一步优化 PostService 类，实现数据库操作与业务逻辑的分离，从而提高系统的结构清晰度和可维护性。

为了提高 PostService 类与数据库操作的解耦度，可以定义一个 PostRepository 接口，该接口声明了操作博客文章数据所需的方法。然后，实现一个 InMemoryPostRepository 类，提供这些方法的具体实现，直接与数据存储交互。通过这种方式，PostService 类仅需与 PostRepository 接口交互，无须了解底层数据存储的具体实现。这增强了模块化，提高了系统的可扩展性和灵活性。

以下是实现的详细步骤。

(1) 创建 PostRepository 接口。

```
import java.util.List;

public interface PostRepository {
    List<Post> getAllPosts();
}
```

(2) 创建 InMemoryPostRepository 类作为 PostRepository 接口的实现。

```
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

public class InMemoryPostRepository implements PostRepository{
    private static final List<Post> posts = new ArrayList<>();

    static {
        // 静态初始化块填充预定义的文章数据
        posts.add(new Post(1L, "欢迎光临我的博客!", "这是第一个帖子",
                "张三", LocalDateTime.of(2029, 7, 1, 12, 0)));
        posts.add(new Post(2L, "如何学习 Spring Boot", "通过 AI 工具学习是个好办法",
                "李四", LocalDateTime.of(2029, 7, 15, 14, 30)));
        // 添加更多预定义文章...

    }

    @Override
    public List<Post> getAllPosts() {
        return posts;
    }
}
```

(3) 修改 PostService 类,创建 InMemoryPostRepository 对象,并使用其方法。

```
import java.util.List;

public class PostService {
    private final PostRepository postRepository = new InMemoryPostRepository();

    public List<Post> getAllPosts() {
        return postRepository.getAllPosts();
    }
}
```

PostService 类已通过 PostRepository 接口与数据访问层交互,实现了数据访问层的抽象和分离。InMemoryPostRepository 类是当前数据访问层的具体实现。然而,PostService 类直接依赖 InMemoryPostRepository 类的具体实现,这限制了其与存储机制的解耦。未来若需切换至数据库等其他存储方式,可能需要修改 PostService 类的函数和逻辑,这会增加维护的复杂性。

通过应用控制反转和依赖注入原则,可以进一步解耦 PostService 类,避免直接绑定到特定实现。

3.2 控制反转与依赖注入

在传统的编程模式中，组件自行负责获取其依赖对象，这增加了它们之间的耦合性，限制了代码的灵活性和可维护性。而控制反转(Inversion of Control，IoC)是将对象的创建和依赖管理的责任从应用程序代码转移到外部容器。在 IoC 框架下，应用程序组件不再自行创建或管理其依赖，而是由容器自动提供所需的依赖项。这种机制降低了组件间的耦合性，简化了测试和维护过程。

IoC 的核心优势在于，它允许容器在运行时自动提供组件所需的依赖，从而让组件专注于实现其业务逻辑，而无须关心依赖的具体实现细节。

Spring Boot 通过依赖注入(Dependency Injection，DI)实现 IoC，允许组件在运行时接收所需依赖，而非自行创建。DI 通过将依赖项的实例直接注入组件中，减少了组件之间的直接依赖。依赖注入通常由一个容器(例如 Spring 框架)管理，该容器根据配置自动装配组件所需的依赖。这种方式让开发者能够集中精力编写业务逻辑，而不必处理依赖对象的生命周期和配置细节。

例如，在上述 PostService 类中使用依赖注入的方式，代码如下：

```
@Service
public class PostService {
    private final PostRepository postRepository;

    public PostService(PostRepository postRepository) {
        this.postRepository = postRepository;
    }
   …
}
```

重构后的 PostService 类采用构造函数注入的方式，接收一个 PostRepository 实例，从而消除了对具体实现的直接依赖。这使得 PostService 类不再需要自行创建 PostRepository 对象，而是接收外部注入的实现，实现了代码的低耦合性。

这种设计能够实现轻松地更换存储机制。例如，若要将存储方式从内存切换到数据库，只需实现一个新的 PostRepository 接口的数据库访问类。在初始化 PostService 时，传入这个新的实现即可。整个过程无须对 PostService 的内部逻辑进行任何更改，体现了高度的灵活性和可维护性。

通过控制反转和依赖注入实现的解耦，提升了代码的灵活性、可维护性和可扩展性。依赖注入可以通过多种方式实现，包括构造函数注入、setter 方法注入、字段注入、注解配置，以及通过 Java 配置类声明和配置 bean。Spring Boot 进一步通过自动配置功能，简化了依赖注入的配置过程。它能够根据项目中添加的依赖自动配置应用程序，减少了显式配置的需求，使得开发更加高效和直观。

1. 构造函数注入

通过构造函数注入依赖项，Spring Boot 确保了对象在创建时即被完全初始化。利用 @Autowired 注解标注构造函数，Spring Boot 自动识别并注入所需的 Bean，简化了依赖注入过程，消除了手动配置依赖的复杂性。

例如,PostService 类通过其构造函数实现依赖注入,示例代码如下:

```
@Service
public class PostService {
    private final PostRepository postRepository;

    public PostService(PostRepository postRepository) {
        this.postRepository = postRepository;
    }
   // 其他业务方法...
}
```

@Service 注解用于标识一个类作为服务层的一部分。应用此注解的类会被 Spring 容器识别为一个 Bean,从而可以利用依赖注入。在 PostService 类上使用@Service 注解,Spring 容器会自动创建其实例,并在其他组件如 PostController 中自动注入,简化了依赖管理并提升了代码的模块化。这种做法使得组件之间的耦合度降低,同时提高了系统的可维护性和可扩展性。

2. Setter 注入

尽管构造函数注入因其确保对象完全初始化而成为推荐做法,但在需要保持类兼容性等特殊情况下,Setter 注入提供了另一种解决方案。这种方法允许在对象创建后配置依赖,提供了额外的灵活性。Setter 注入通过类的公共 setter 方法在对象实例化后设置依赖项,增加了配置的灵活性。在 Spring 框架中,可以使用@Autowired 注解来标记 setter 方法,这样 Spring 容器就会自动注入相应的依赖项。

例如,PostService 类通过 Setter 注入实现依赖注入,示例代码如下:

```
public class PostService {
    private PostRepository postRepository;

    @Autowired
    public void setPostRepository(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    // 其他业务方法...
}
```

在上述示例中,PostService 类使用@Autowired 注解的 setter 方法来接收 PostRepository 的实例。Spring 容器负责在适当的时候调用这个方法并注入依赖项。

3. 字段注入

字段注入是 Spring 框架中实现依赖注入的另一种方式,它通过直接在类的字段上使用注解来完成。这种方式简便直观,但相比构造函数注入和 Setter 注入,它有其特定的使用场景和限制。它适用于依赖关系简单、不需要复杂初始化逻辑的场景,提供了代码编写上的便捷性。然而,字段注入也有其局限性,如不适用于 final 字段,可能隐藏依赖关系,影响代码的清晰度和测试性,以及在多线程环境中可能需要额外的线程安全措施。

例如,PostService 类通过字段注入实现依赖注入,示例代码如下:

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    // ...
}
```

在上述示例中，PostService 类中的 postRepository 字段通过@Autowired 注解直接注入，Spring 容器负责提供 PostRepository 的实例。这种方式虽然方便，但可能导致代码可读性和可维护性的下降，因为这种隐式的方式可能会隐藏类之间的依赖关系。为了提升代码的清晰度和易于维护，通常建议优先采用构造函数注入，以明确地表示依赖。

【例 3-3】 使用依赖注入的方式优化例 3-2，使代码更加灵活、可维护和可扩展。

要将 PostService 修改为使用依赖注入，首先确保 InMemoryPostRepository 类被 Spring 框架管理，这可以通过在类上添加@Repository 注解来实现。接下来，从 PostService 类中移除对 InMemoryPostRepository 类的直接实例化，改为通过构造函数注入该依赖项。以下是更改后的示例代码：

```
@Repository
public class InMemoryPostRepository implements PostRepository{
// 实现细节...
}
```

在这个示例中，InMemoryPostRepository 类通过@Repository 注解被标识为 Spring 管理的 Bean。PostService 类采用了构造函数注入的方式，以注入 PostRepository 接口的实现。以下是依赖注入优化后的 PostService 类代码示例：

```
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class PostService {
   private final PostRepository postRepository;

   public PostService(PostRepository postRepository){
       this.postRepository = postRepository;
   }

   public List<Post> getAllPosts() {
       return postRepository.getAllPosts();
   }
}
```

通过这种方式，PostService 类不再需要自己创建 PostRepository 实例，而是依赖 Spring 容器提供的实例，实现依赖的自动注入和松耦合。同理，PostController 类也可以使用依赖注入的方式来接收 PostService 类的实例。修改后的代码如下：

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService postService;

    @Autowired
    public PostController(PostService postService) {
        this.postService = postService;
    }
    @GetMapping
    public ResponseEntity<List<Post>> getAllPosts() {
        // 返回文章列表
        return ResponseEntity.ok()
                .body(postService.getAllPosts());
    }
}
```

这种方式的修改,实现了依赖的自动注入和代码解耦,使得代码更易于测试和维护,同时也更好地遵循了面向接口编程的原则。

3.3 自动配置

Spring Boot 自动配置极大地简化了 Spring 应用的开发流程,它让应用程序在几乎零配置的情况下就能正确地运行起来。它通过自动检测项目中的依赖项、智能地应用条件配置 Bean,以及提供直观的配置选项,帮助开发者快速构建功能完善的应用,同时保持了应对个性化需求的灵活性。这种机制允许开发者将重点放在业务逻辑上,而不是深陷底层配置的细节,显著提高了开发效率。

在传统的 Spring 框架中,创建和管理 Bean 通常涉及编写大量的 XML 配置或 Java 配置类,定义 Bean 的生成、依赖和生命周期管理,这在大型项目中可能会变得相当复杂。Spring Boot 通过其自动配置特性,能够根据添加到项目中的依赖自动生成所需的 Bean 配置,减少了手动编码的工作量,提升了开发效率和便捷性。

例如,在使用数据库时,传统 Spring 应用通常需要手动配置数据源、事务管理器等核心组件。以下是一个传统 Spring 应用中配置数据源和事务管理器的简化示例,代码如下:

```
@Configuration
public class DatabaseConfig {
```

```
    @Value("${spring.datasource.url}")
    private String url;

    @Value("${spring.datasource.username}")
    private String username;

    @Value("${spring.datasource.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName("com.mysql.jdbc.Driver");
        dataSource.setUrl(url);
        dataSource.setUsername(username);
        dataSource.setPassword(password);
        return dataSource;
    }

    @Bean
    public PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

这段代码是一个 Spring 应用的传统 Java 配置示例，它通过@Configuration 注解 DatabaseConfig 类手动配置了数据库连接和事务管理。使用@Value 注解从配置文件中注入数据库连接参数，并创建了 DataSource 和 PlatformTransactionManager 类的 Bean，展示了 Spring 的基于 Java 的配置方法。

Spring Boot 通过自动配置机制极大地简化了传统 Spring 应用中的配置流程。开发者只需在 application.properties 文件中指定数据库连接参数，Spring Boot 便能自动识别并配置数据源和事务管理器，无须编写烦琐的配置类。

例如，通过在 application.properties 添加如下配置：

```
spring.datasource.url = jdbc:mysql://localhost:3306/mydb
spring.datasource.username = myuser
spring.datasource.password = mypassword
spring.jpa.hibernate.ddl - auto = update
```

一旦项目引入了相应的依赖(如 spring-boot-starter-data-jpa)，Spring Boot 将根据这些配置自动设置数据库连接和 JPA 属性。这种方式不仅减少了手动配置的工作量，而且提高了开发效率，使开发者能够专注于业务逻辑的实现。如果需要对数据库连接池或 JPA 进行更细致的定制，Spring Boot 同样提供了简便的配置方式。开发者可以直接在配置文件中添加或修改相关属性，而无须更改 Java 代码，这体现了 Spring Boot 在配置灵活性和便捷性方面的优势。

通过示例可知，Spring Boot 如何使开发者能够快速构建具有数据库功能的 Web 应用程序，而无须深入复杂的数据库配置。这种方法显著提升了开发效率并优化了开发体验。

3.4 依赖管理

在软件开发中,“依赖”指的是项目或模块运行时所依赖的外部库、框架或服务,它们提供基础功能,促进代码复用,降低开发成本。依赖的合理运用可以加速开发进程、保障系统稳定性、满足特定的功能需求,并推动技术栈的统一和进步,从而构建出高效、高质量的应用程序。

有效的依赖管理是项目成功的关键,它确保了版本间的兼容性,简化了配置流程,自动化了依赖的检索与更新,并减少了构建失败的风险,提升了开发效率和项目质量。Spring Boot 项目通过使用 Starter 依赖和父 POM 管理,实现了便捷的依赖管理方式。Starter 依赖自动整合了常用的库和配置,提供了经过验证的依赖版本组合,极大地简化了手动依赖管理的工作量,确保了项目构建的一致性和效率,让开发者可以迅速开始项目开发。

3.4.1 Starter 依赖

Spring Boot 的 Starter 依赖提供了一种高效的依赖管理方法,它是一组预先配置的库集合,专门针对特定技术或功能,如 Web 开发、数据库操作和安全性。这些依赖以统一的命名格式 spring-boot-starter-* 表示,例如 spring-boot-starter-web 和 spring-boot-starter-data-jpa,它们通过简化依赖的添加和管理,加快了项目的构建和功能的集成,显著提高了开发效率。

每个 Starter 可以视为一个功能完备的工具箱,包含了实现特定功能所需的全部依赖。以 spring-boot-starter-web 为例,它不仅提供了 Spring MVC 框架,还内置了 Tomcat 内嵌服务器和 Jackson 库来处理 JSON 数据。开发者只需在项目中加入这个 Starter 依赖,Spring Boot 便会自动配置所需的 Web 组件,免去了手动配置的烦琐。这种自动化配置极大地减轻了开发者的工作负担,使得他们能更专注于核心业务逻辑的实现。

常用 Starter 如表 3-1 所示。

表 3-1 常用 Starter

名 称	作 用
spring-boot-starter-web	包含 Spring MVC 和内嵌的 Web 服务器,适用于构建 Web 应用程序
spring-boot-starter-data-jpa	集成 Spring Data JPA,用于简化数据持久化
spring-boot-starter-security	集成 Spring Security,用于处理应用程序的安全性
spring-boot-starter-data-redis	用于集成 Redis 数据库,包含 Spring Data Redis 等相关依赖项
spring-boot-starter-cache	用于集成缓存功能,包含 Ehcache、Redis 等相关依赖项
spring-boot-starter-logging	用于日志记录,包含 Logback、Log4j2 等相关依赖项

此外,Spring Boot 框架也支持开发者根据特定需求定制 Starter 模块,以封装和复用一组特定功能的配置和依赖,读者有兴趣可以自行了解。

3.4.2 父 POM 管理

父 POM 管理是 Maven 中用于集中化项目配置的功能,它允许在多模块项目中通过一个共享的父 POM(位于项目根目录的 pom.xml)来统一定义构建配置和依赖。子模块通过继承此父

POM，能够自动采用其中定义的配置，无须在各自的 pom.xml 中重复设置。这种方法简化了项目管理，确保了依赖版本的统一，并使所有子模块都遵循统一的构建规范。

Spring Boot 提供了 spring-boot-starter-parent 作为官方父 POM，它包含了 Spring Boot 应用的最佳实践配置，简化了 Spring Boot 项目的构建配置。父 POM 管理是大型项目和企业级应用中常见的做法，它提高了项目的可维护性和可扩展性，同时降低了管理成本。

以下是一个简化的例子，展示了如何在 Maven 项目中使用 spring-boot-starter-parent 作为父 POM，并添加了必要的 Starter 依赖。

Maven 项目的 pom.xml 配置示例：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- 继承 Spring Boot 的父级项目 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.0</version> <!-- 使用具体的版本号 -->
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <groupId>com.example</groupId>
    <artifactId>my-spring-boot-app</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <dependencies>
        <!-- 添加 Spring Boot Web Starter 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- 如需数据库访问，可添加 Spring Data JPA Starter -->
        <!-- 注意：这也会自动引入相应的数据库驱动依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
    </dependencies>

    <!-- 默认的构建插件和生命周期配置已由父级项目提供 -->
</project>
```

在这个配置中，通过继承 spring-boot-starter-parent，项目自动获得了 Spring Boot 推荐的依赖版本和插件配置。开发者无须手动指定每个依赖的版本，因为父 POM 已经管理了这些信息。这种方式简化了依赖管理，加快了项目设置的速度，并确保了项目的稳定性和一致性。

3.5 综合应用：博客项目的三层架构重构

3.5.1 案例描述

在第 2 章的综合案例中，所有业务逻辑都集中在控制器中实现，这种方法适用于小型项目。然而，随着项目复杂度和规模的增长，这种集中式的做法可能会导致控制器层变得过于臃肿和混乱，影响项目的可维护性和可扩展性。为了解决这个问题，本案例采用三层架构模式进行了重构，将业务逻辑从控制器中分离，并通过依赖注入技术实现了代码的解耦，从而提高了代码的清晰度和可维护性。

3.5.2 案例实现

将代码的职责明确划分为数据访问层、业务逻辑层和表现层，这样的分离目的是增强应用的可维护性、可测试性和可扩展性。以下是一个简化的示例。

1. 数据访问层

定义一个接口及其实现类来管理博客文章的数据交互，这通常涉及数据库操作。接口声明了数据访问的方法，而实现类则提供了这些方法的具体实现，与数据库进行通信。

(1) 定义接口。

PostRepository 接口定义了一组操作博客文章数据的方法，这些方法通常用于与数据存储(如数据库)进行交互。接口包含以下方法。

getAllPosts()：返回一个包含所有博客文章的列表。

getPostById(Long id)：根据提供的文章 ID 返回一个博客文章对象。

createPost(Post post)：创建一个新的博客文章，并将其添加到数据存储中。

deletePost(Long id)：根据提供的文章 ID 从数据存储中删除博客文章。

这个接口为数据访问层提供了一个抽象层，允许不同的实现类以不同的方式(例如，使用不同类型的数据库)来实现这些数据操作，同时保持业务逻辑层的一致性。示例代码如下：

```
import java.util.List;

public interface PostRepository {
    List<Post> getAllPosts();
    Post getPostById(Long id);
    Post createPost(Post post);
    boolean deletePost(Long id);
}
```

(2) 接口的实现。

InMemoryPostRepository 类是 PostRepository 接口的一个实现，它提供了一个简单的内存数据存储来管理博客文章。这个实现使用了一个 static 的 ArrayList<Post>来模拟数据库中的数据表。示例代码如下：

```
import org.springframework.stereotype.Repository;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

@Repository
public class InMemoryPostRepository implements PostRepository{
    private static final List<Post> posts = new ArrayList<>();

    static {
        // 静态初始化块填充预定义的文章数据
        posts.add(new Post(1L, "欢迎光临我的博客!", "这是第一个帖子",
                "张三", LocalDateTime.of(2029, 7, 1, 12, 0)));
        posts.add(new Post(2L, "如何学习 Spring Boot", "通过 AI 工具学习是个好办法",
                "李四", LocalDateTime.of(2029, 7, 15, 14, 30)));
        // 添加更多预定义文章...

    }

    @Override
    public List<Post> getAllPosts() {
        return posts;
    }

    @Override
    public Post getPostById(Long id) {
        return posts.stream()
                .filter(post -> post.getId().equals(id))
                .findFirst()
                .orElse(null);
    }

    @Override
    public Post createPost(Post post) {
        // 生成新的 ID,假设没有 ID 冲突
        long newId = posts.stream().mapToLong(Post::getId).max().orElse(0L) + 1;
        post.setId(newId);
        posts.add(post);
        return post;
    }



    @Override
    public boolean deletePost(Long id) {
        Post existingPost = getPostById(id);
        if (existingPost != null) {
            posts.remove(existingPost);
            return true;
```

```
        }
        return false;
    }
}
```

这个实现没有使用真正的数据库,而是在内存中进行操作,适用于测试或小型应用。在实际应用中,可能会使用JPA、MyBatis或其他ORM(对象关系映射)工具来实现数据访问层的逻辑,与数据库进行交互,这在第4章中会详细介绍。

2. 业务逻辑层

PostService类是一个服务组件,用于封装与博客文章相关的业务逻辑。它通过依赖注入获得数据访问层的PostRepository实例,并提供了一系列方法来执行获取、创建、删除文章的操作。示例代码如下:

```
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class PostService {
    private final PostRepository postRepository;

    public PostService(PostRepository postRepository){
        this.postRepository = postRepository;
    }

    public List<Post> getAllPosts() {
        return postRepository.getAllPosts();
    }

    public Post getPostById(Long id) {
        return postRepository.getPostById(id);
    }

    public Post createPost(Post post) {
        return postRepository.createPost(post);
    }

    public boolean deletePost(Long id) {
        return postRepository.deletePost(id);
    }
}
```

3. 表现层

表现层是应用程序中与用户直接交互的部分,主要负责处理HTTP请求、生成响应,以及展示用户界面。PostController类作为表现层的组件,通过依赖PostService对象来处理业务逻辑,避免了直接操作数据的细节,从而简化了控制器的职责并实现了更清晰的职责划分。这种设计使得PostController类专注于用户交互,而将数据处理委托给服务层。示例代码如下:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService postService;

    @Autowired
    public PostController(PostService postService) {
        this.postService = postService;
    }
    @GetMapping
    public ResponseEntity<List<Post>> getAllPosts() {
        // 返回文章列表
        return ResponseEntity.ok()
                .body(postService.getAllPosts());
    }

    @GetMapping("/{postId}")
    public ResponseEntity<Post> getPostById(@PathVariable Long postId) {
        Post post = postService.getPostById(postId);

        if (post == null) {
            // 如果找不到对应的博客,返回 404 Not Found 响应
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok().body(post);
    }



    @PostMapping
    public ResponseEntity<Post> createPost(@RequestBody Post newPost) {

        return ResponseEntity.ok()
                .body(postService.createPost(newPost));
    }



    @DeleteMapping("/{postId}")
    public ResponseEntity<Void> deletePost(@PathVariable Long postId) {
        boolean postDeleted = postService.deletePost(postId);
        if (postDeleted) {
            return ResponseEntity.noContent().build(); // HTTP 204 No Content,表示删除成功
```

```
            } else {
                return ResponseEntity.notFound().build();
            }
        }
    }
```

其中：

getAllPosts()方法处理 GET 请求，返回所有文章的列表。

getPostById(@PathVariable Long postId)方法处理带有特定文章 ID 的 GET 请求，返回单个文章或 404 错误。

createPost(@RequestBody Post newPost)方法处理 POST 请求，创建并返回新文章。

deletePost(@PathVariable Long postId)方法处理 DELETE 请求，根据文章 ID 删除文章，并返回 204 状态或 404 错误。

3.5.3 案例总结

通过分层架构设计，各个层次的职责得以明确。表现层专注于处理 HTTP 请求和响应，确保与用户的交互顺畅；业务逻辑层则承担起执行核心业务规则的任务；而数据访问层专门负责与数据存储进行交互。通过依赖注入，各层之间的耦合度得以降低，这不仅提升了代码的可测试性，也增强了系统的可维护性。在后续章节中，读者将会看到如何轻松地将数据源切换到实际的数据库，进一步提升系统的实用性。

习题 3

1. 在 Spring Boot 中实现字段级别的依赖注入的方法是(　　)。
   A. 使用@Autowired 注解在字段上　　B. 使用@Resource 注解在字段上
   C. 在构造器中通过参数注入　　D. 都不对
2. 以下不是 Spring 进行依赖注入的方式是(　　)。
   A. 使用@Autowired 注解　　B. 使用构造函数注入
   C. 使用 setter 方法注入　　D. 使用 new 关键字实例化对象
3. Spring Boot Starter 的作用是(　　)。
   A. 提供快速集成常用库的依赖集合　　B. 用于编写微服务
   C. 用于自动化配置 Spring 框架　　D. 用于编写单元测试
4. Spring Boot 的 Starter 父 POM 是(　　)。
   A. spring-boot-starter-parent　　B. spring-boot-starter
   C. spring-boot-autoconfigure　　D. spring-framework-bom
5. 使用 spring-boot-starter-web 是为了引入(　　)依赖。
   A. Spring MVC 和 Tomcat　　B. Spring Data JPA
   C. Spring WebSocket　　D. Spring Security
6. 实现更新已存在的博客文章功能。